Architecting Self-adaptive Software Systems

Huuhtanen, Anni

Springer Nature Switzerland
2018

Huuhtanen , A , Mäkitalo , N K & Mikkonen , T J 2018 , Architecting Self-adaptive Software Systems . in C Pautasso , F Sánchez-Figueroa , K Systä & J M M Rodrigue (eds) , Current Trends in Web Engineering : ICWE 2018 International Workshops, MATWEP, EnWot, KD-WEB, WEOD, TourismKG, Cáceres, Spain, June 5, 2018, Revised Selected Papers . Lecture Notes in Computer Science , vol. 11153 , Springer Nature Switzerland , Cham , pp. 59-70 , 2nd International Workshop on Engineering the Web of Things , Cáceres , Spain , 05/06/2018 . < https://link.springer.com/chapter/10.1007/978-3-030-03056-8_6 >

http://hdl.handle.net/10138/321582

acceptedVersion

Architecting Self-Adaptive Software Systems

Anni Huuhtanen, Niko Mäkitalo, and Tommi Mikkonen

Department of Computer Science
University of Helsinki
Helsinki, Finland
{anni.huuhtanen,niko.makitalo,tommi.mikkonen}@helsinki.fi

Abstract. A growing number of software systems operate in uncertain environments. They benefit from an ability to autonomously adapt to changes during runtime without suffering from a lowered quality of service. Several different architectural approaches to self-adaptive software exist with their sources of inspiration varying from psychology to mathematics. In this literature survey, we study and evaluate four types of approaches: architecture-based, control-based, learning-based and awareness-based approaches. Our aim is to clarify whether a unified, general approach to computational self-adaptivity is possible and what it could look like. We conclude that a general solution should combine aspects of all of the studied approaches.

Keywords: self-adaptivity · software architectures · autonomous systems · Internet of Things · IoT · Web of Things · WoT.

1 Introduction

Software systems are becoming increasingly complex as the trend for distributed, heterogeneous and large-scale architectures grows. This is especially visible in the Internet of Things (IoT), and the Web of Things (WoT) systems, which may consist of thousands of parts or subsystems, each processing information locally and exchanging it without any centralized controller. Each subsystem can even be owned by a different agent or organization. Despite possibly having diverse properties and behaviors, they must be able to interact with each other.

In addition, requirements of various stakeholders such as cost, performance and safety can be at odds, meaning that a system must make certain trade-offs. Traditionally, these trade-offs have been analyzed during the design period of software and addressed as various requirements. However, in more and more cases, the runtime operation of the system results from complex interactions between several moving parts and therefore tends to be uncertain at any given time. This uncertainty means that the same behavior does not always lead to the same outcome or that, for example, sometimes a particular subsystem may be accessible to others and sometimes not. The operation of such an uncertain system after deployment is difficult to predict and infeasible to manually control, leading to a need for self-adaptivity.

Self-adaptivity in software systems means the ability of a system to adapt to changes in its environment during runtime in an independent manner without downtime or penalties [1]. A self-adaptive system is equipped with capabilities to observe itself and its environment in order to make autonomous decisions on how to behave. It can configure, optimize, repair, and protect itself based on what it observes. An observation could be, for example, input from a user or another system, or an anomaly in its own internal state. Self-adaptive mechanisms range from simple conditional expressions where the outcome changes program behavior to more complicated machine learning techniques where behavioral change occurs as a result of learning new knowledge.

A need for computational self-adaptivity arose in the early 2000s when the maintenance and operation of computing systems was still done manually by human administrators [2]. Typical maintenance tasks included installing, configuring and optimizing software and hardware. As the amount of computers and interconnections in computing systems started to grow, manual management proved to be cumbersome and methods to deal with this increasing complexity were needed. The trend for distributed, diverse and large computing systems has continued to this day. Many modern software systems, for example the areas of IoT and WoT, benefit from or even require self-adaptivity in many ways. An example is a cloud-based software service that dynamically allocates resources to a number of applications. In the case of a single application, the service must optimize both quality of service (QoS) and a cost objective [3]. Since the same infrastructure is shared by several applications, it must additionally offer good QoS to all applications. Uncertainty of the runtime environment and possibly conflicting objectives require the service to have self-adaptive properties.

Different architectural approaches to computational self-adaptivity have been developed over time. It can be seen that the approaches build on each other in a wave-like manner [2]. At the moment, the focus has started to move from traditional architecture-based approaches to more mathematical, control-based approaches. Recent attempts to incorporate self-adaptivity into software have also utilized machine learning techniques as well as taken inspiration from psychological theories and concepts such as self-awareness.

Since multiple approaches to computational self-adaptivity exist, it can be difficult to get a holistic understanding of the field. There has also been little comparison between the approaches. A comprehensive theoretical foundation of computational self-adaptivity is still missing and the path to achieve it seems unclear.

In this paper, we examine and evaluate different architectural approaches to developing self-adaptive software based on existing literature. The main purpose of the study is to clarify what is required for a more unified approach to computational self-adaptivity. Using the Goal-Question-Metric approach [4], our goal is formulated as follows:

- **Purpose**: Characterize and evaluate ...
- **Issue**: ... different architectural approaches to self-adaptive software ...
- **Objective**: ... in order to move towards a unified theory ...

– **Viewpoint**: ... from the standpoint of a researcher.

The precise research questions are the following:

– **RQ1**: What kind of architectural approaches to self-adaptive software exist?
– **RQ2**: What are the benefits and problems of different approaches?

The rest of this paper is structured as follows. In Section 2, we explain how we searched for articles for our survey and justify their selection. In Section 3, we examine different self-adaptive approaches and provide answers to our research questions. In Section 4, we discuss another recent literature survey. Finally, in Section 5 we draw some final conclusions.

2 Methods

First, we defined the search string to be used for automatic search. While doing initial research on the problem and formulating the research questions, we made a few searches using the single keywords self-adaptive and software. However, it became evident that multiple synonyms for these words existed in literature. For example, self-adaptivity and autonomy were often used interchangeably. Therefore, we defined the final search string as follows.

(self-adaptive OR self-adaptivity OR self-managing OR autonomic OR autonomous OR autonomy) AND (software OR framework OR architecture)

Then, we performed searches using the search string on IEEE Xplore Digital Library (*https://ieeexplore.ieee.org/*), ACM Digital Library (*https://dl.acm.org/*), and Google Scholar (*https://scholar.google.fi/*). We applied all keywords to the title field only, and searched only for articles that were published between 2016 and 2018. The searches resulted in 184 articles on IEEE Xplore, 277 articles on ACM Digital Library, and 653 articles on Google Scholar.

From the results, we chose the most recent literature survey on computational self-adaptivity as a starting point for snowballing. Using the snowballing method, we checked sources used by the survey and then looked at other articles where those sources had been cited. From these, we decided to select articles that present not only general theoretical approaches but also specific models so that we could better evaluate them. Finally, this lead to 10 articles for our survey, from which five articles present a distinct architectural model for computational self-adaptivity, and the rest are about general categories or approaches which these models can be classified into.

3 Results

The approaches we studied can roughly be classified into architecture-based, control-based, awareness-based and learning-based approaches, some having aspects of more than one class. Table 1 summarizes the findings.

Table 1. Identified self-adaptive approaches.

Approach	Examples	Inspiration	Emphasis	Idea
Architecture-based	MAPE-K, 3LA	Software engineering	Basic building blocks of self-adaptive systems	The system is equipped with a self-adaptive architecture following known principles of software engineering
Control-based	SimCA*	Mathematics	Formal guarantees for self-adaptivity	The self-adaptive process is guaranteed to adhere to the requirements of the system with the use of control theory
Learning-based	FUSION	Machine learning, software engineering	Learning self-adaptive logic through experience	The system learns the optimal self-adaptive behavior at runtime using incremental learning algorithms
Awareness-based	SAA	Psychology, machine learning, software engineering	Enhanced building blocks of self-adaptivity, enhanced learning	The system is equipped with an architecture that allows self-awareness, learning occurs on multiple levels, predicting future possible

3.1 Architecture-based approaches

Architecture-based approaches emphasize the role of architecture in designing self-adaptive software systems [1]. Instead of focusing on how to encode self-adaptivity into source code, the emphasis is on the structure and interaction of higher level software components and how they could implement self-adaptive behavior. Basic principles of software architecture design such as separation of concern and abstraction are applied. For example, components implementing self-adaptivity are separated from components implementing normal functionality, and each self-adaptive component has its own responsibility. In addition, the details of each component are hidden behind an interface. Components and their connectors can be modified, added and removed at runtime as well as organized in different ways.

MAPE-K is one of the first self-adaptive models that have been developed [5]. In the model, a software system is coupled with a separate managing component that implements four functions: Monitor, Analyze, Plan, and Execute. All functions have access to a shared Knowledge base that includes high-level goals given by the system administrator as well as knowledge gathered during runtime. The Monitor component gathers information about the system and its environment, updating the Knowledge component accordingly. The Analyze component determines whether adaptation actions are required based on the knowledge. If they are, the Plan component creates a plan of action and gives it

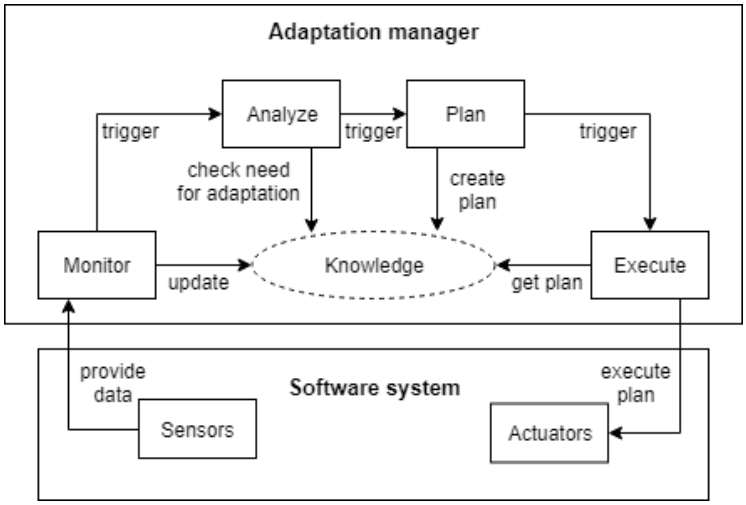

Fig. 1. The MAPE-K model [5].

to the Execution component that implements it. The MAPE-K model is shown in Figure 1.

The problem with the MAPE-K model is that it specifies the different functions of a self-adaptive system on a very high level [5]. It is also quite simple and unprincipled as a software architecture. However, it gives a general overview of the basic functions needed in almost every self-adaptive system and works as an inspiration for more low-level architectures.

3LA (Three-Layer Architecture) consists of three layers: Component Control, Change Management and Goal Management [6]. The bottom layer, Component Control, represents the software system itself and is monitored and reacted to by the middle layer, Change Management. The middle layer implements predefined self-adaptation models or plans and can, for example, remove and add components as well as modify their parameters. If no suitable plan exist, the middle layer triggers the top layer, Goal Management. This layer handles the high-level goals of the system and generates new plans for the middle layer. The 3LA model is shown in Figure 2.

3LA is slightly more precisely defined than MAPE-K because of its layered structure, but it has the same problem of being simple and high-level [6]. It is unclear what the different layers exactly contain, for example.

3.2 Control-based approaches

In some software systems, self-adaptivity must occur according to precise requirements and despite restrictions such as time or memory. However, it can be difficult to guarantee these self-adaptation goals are met in architecture-based

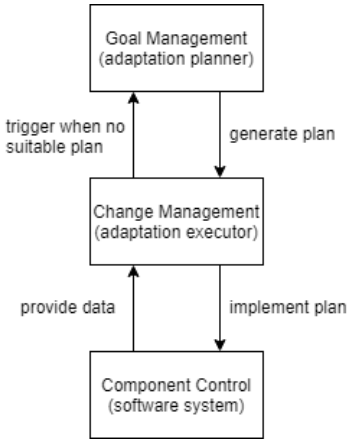

Fig. 2. The 3LA model [6].

approaches where traditional verification techniques, for example tests and model checking, are used [7]. The idea with control-theoretical approaches is that formal guarantees for self-adaptation can be provided. Whereas architecture-based approaches are inspired by principles of software engineering, the foundation of control-based approaches is control theory from mathematics. Control theory has traditionally been used to implement self-adaptivity at the level of hardware, for example CPU and memory. Lately, it has also been applied to software adaptation.

Control theory focuses on modeling software behavior instead of components [7]. This behavior involves two schemes, feedback control and feedforward control. In feedback control, when the system goes from functioning normally (a steady state) to experiencing internal or external changes or disturbances (a transient state), the output of the system is measured and compared to the self-adaptation goal (setpoint) such as response time. The error between the output and the setpoint determines a control signal that adapts the software system in a way that decreases the error. The system then enters back to a steady state. Feedforward control on the other hand calculates a different control signal based on the setpoint and the values of disturbances instead. It can be said that feedback control selects the initial adaptation strategy, and feedforward control refines it and ensures it is followed. Both schemes can implement different control strategies: in optimal control, the control signal that minimizes a certain cost function is selected while in state feedback, the signal is computed based on state information. Feedback and feedforward controllers may also be composed of multiple hierarchically organized sub-controllers.

SimCA* (Simplex Control Adaptation) is a control-theoretical model that supports three types of goals or requirements [8]. Setpoints (S-reqs) are simple values the system should achieve. Optimizable values (O-reqs) are functions the system should optimize. Threshold values (T-reqs) have to stay below or above a

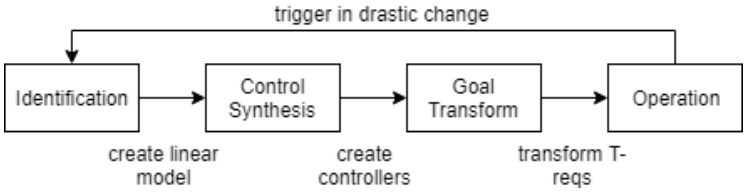

Fig. 3. The SimCA* model [8].

threshold. SimCA* works in four phases: Identification, Control Synthesis, Goal Transformation and Operation. In Identification, SimCA* builds a linear model of each goal by measuring the dependency between different goal values (control signals) and the system output in the form of a coefficient value. In Control Synthesis, a controller function is built for each of these models. The function calculates the control signal based on several values such as the previous control signal, coefficient, and error between the previous control signal and system output. The controller also includes mechanisms for adjusting the linear model and re-triggering Identification during sudden runtime changes. In Goal Transformation, all threshold goals are adjusted in a way that allows other goals to be satisfied as well. In Operation, the controllers output control signals and the final combined control signal is calculated based on them and possible optimization goals. The SimCA* model is shown in Figure 3.

Although SimCA* is a more formal and mathematical model than MAPE-K, for example, its use in practice does not require a mathematical background which might still make it attractive to a regular software engineer [8]. The problem with SimCA* is that several drastic changes during runtime result in frequent re-identification which is costly and may hinder performance. In addition, the model cannot produce a satisfiable outcome in case of conflicting requirements. Unlike previous control-theoretical approaches, SimCA* is argued to be better suitable for multiple domains and problems. However, the framework has only been tested to work in a couple of scenarios and it is not certain if it generalizes well to different types of problems. Another problem with SimCA* and control-theoretical approaches in general is that not all requirements can easily be transformed into numerical form.

3.3 Learning-based approaches

Learning-based approaches utilize statistical and machine learning techniques. They often use a MAPE-K-like architecture, but instead of equipping the software system with an analytical, static model that controls self-adaptation during runtime, learning-based approaches allow the system to learn the optimal adaptation logic through experience.

FUSION (FeatUre-oriented Self-adaptatION) is a learning-based adaptive framework that is based on features [9]. They are abstractions of the system's

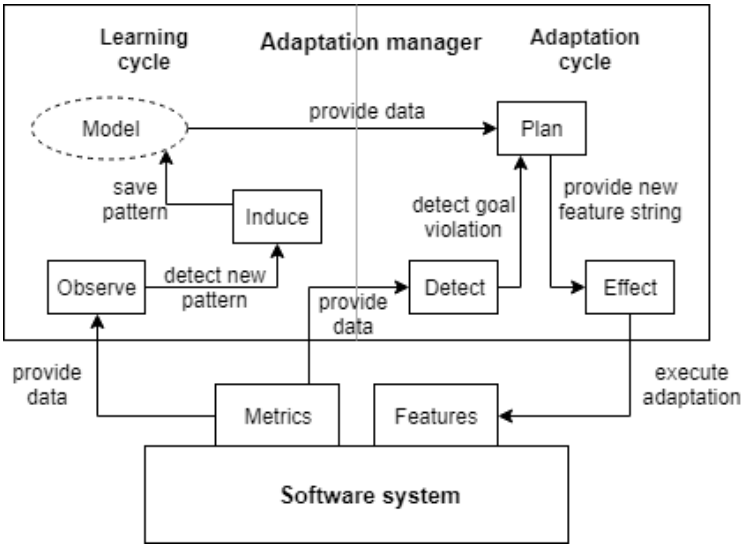

Fig. 4. The FUSION model [9].

abilities and the subjects or units of adaptation. Depending on the application, they may correspond to, for example, a set of services, rules or properties. At any given time, the software system has certain features activated and others deactivated. This state is captured in a feature selection string where activated features are set to 1 and deactivated features to 0. The adaptation of the system is modeled as transitions between these feature selection strings.

FUSION involves both an adaptation cycle and a learning cycle as well as an internal model [9]. The adaptation cycle consists of three components, Detect, Plan and Effect, that are similar to the components in MAPE-K. Detect analyzes metrics gathered from the system and detects if an adaptation goal has been violated. If it has, Plan uses the model to determine which system features should be activated or deactivated, i.e. creates the next feature selection string. Finally, Effect implements the transition from the current feature selection string to the next in a way that ensures system stability. Meanwhile, the learning cycle observes relationships between the system metrics and implemented transitions and looks for unexpected patterns, in which case the model (used by Plan) adjusts to the new pattern, improving future adaptation. The FUSION framework is shown in Figure 4.

FUSION is argued to be efficient, easy to use and accurate [9]. The abstraction achieved with features improves the generality of the framework. The problem with FUSION is that the incremental learning brings some overhead to the system. In addition, the learning cycle cannot predict abnormal patterns in an opportunistic manner. There has also been little testing of FUSION in real-world scenarios.

3.4 Awareness-based approaches

Awareness-based approaches utilize aspects of both architecture- and learning-based approaches. Self-adaptivity is related to the concepts of self-awareness and self-expression that originate from psychology [10]. Lewis et al. have developed five-level model of consciousness (stimulus-awareness, interaction-awareness, time-awareness, goal-awareness, meta-awareness) [11, 10] which originate from Neisser's levels of human self-awareness [12].

Computational self-awareness means the ability of a system to know about its own internal state as well as how it is perceived by other systems in its environment. A stimulus-aware system can respond to internal and external stimuli but does not know the difference between different types of stimuli. An interaction-aware system knows that its actions cause effects in the environment because of feedback loops. A time-aware system has knowledge about the past and can predict the future. A goal-aware system is aware of runtime constraints and objectives that are not implicitly present in its design. A meta-self-aware system is aware of being aware and is able to modify all lower layers if necessary. Computational self-expression on the other hand is the behavior resulting from knowledge gained through self-awareness. In the context of self-adaptivity, we can say that self-awareness concerns information processing and that self-expression consists of the resulting adaptive actions. Therefore, an advanced self-adaptive system is both self-aware and self-expressive.

In many modern software systems, the behavior of individual system components is not as important or interesting as the collective behavior emerging from the interaction of multiple components [11]. One observation from biological systems is that self-awareness seems to be an emergent property of distributed information processing, meaning that all components of a system only have access to local knowledge instead of a global, centralized knowledge base. This indicates that a software system could perhaps be more self-adaptive and less fragile with a similar decentralized architecture.

SAA, an adaptive architecture based on self-awareness, has been introduced [11]. In this architecture, an individual software application includes separate self-awareness and self-expression processes. The self-awareness process has access to internal models and is further divided into sub processes corresponding to the different levels (goal-awareness, stimulus-awareness and so on). Each process analyzes information gathered with internal and external sensors and possibly updates the internal models using its own incremental learning algorithm. The goal-aware sub process also has access to design-time goals, and the meta-self-aware sub process can change the goals as well as the learning algorithms of other sub processes. The models are exposed to the self-expression process that chooses actions based on them. The self-awareness process is also able to monitor what actions are taken. SAA is illustrated in Figure 5.

As in FUSION, the knowledge in SAA is modified during runtime, meaning the adaptation logic itself can change, and benefits of architecture-based approaches are achieved [11]. Unlike FUSION, the self-aware architecture specifies different types of knowledge representing different levels of self-awareness

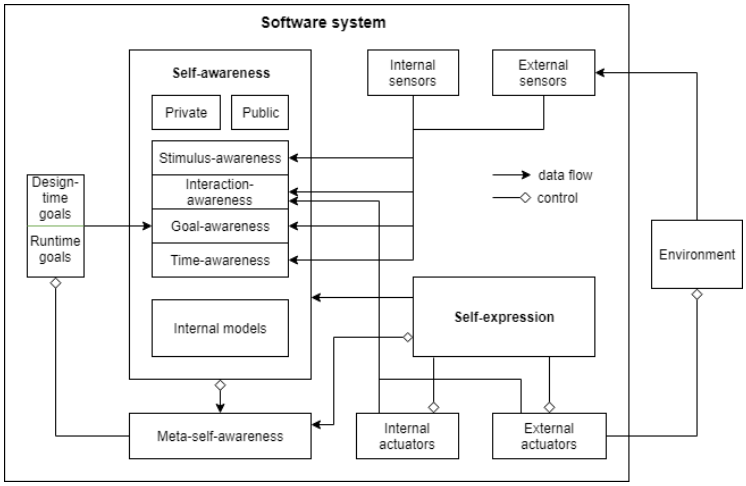

Fig. 5. The SAA model [11].

instead of only one type of knowledge that is used as the basis for adaptation. In addition, the time-aware component allows prediction of future events. The overhead brought by incremental learning and the lack of practical evaluation are problems in SAA as well.

4 Related work

The work by Weyns et al. provides a perspective on how different self-adaptive approaches have developed over time [2]. The history of research on self-adaptivity is structured into six waves that reflect different research trends in the field. Each wave has usually attempted to solve a particular problem that has emerged during the previous waves.

The first wave defined the basic aspects of computational self-adaptivity and focused on the automation of typical system administration tasks such as installing and configuring software systems [2]. The simple self-adaptive model, MAPE-K, was the main contribution of the first wave.

The second wave attempted to specify in more detail how to actually engineer self-adaptive systems [2]. The principles of software architecture design such as abstraction and separation of concern were applied. At this time, the concept of architecture-based self-adaptivity was born and MAPE-K evolved into 3LA.

While the first two waves defined the fundamental principles of self-adaptivity and offered a few imprecise but general architectures, the third wave focused on concrete self-adaptation mechanisms [2]. Techniques of model-driven engineering that traditionally were applied at design time were now extended to runtime environments. Runtime models of a software system represent the system at an abstract level, and the idea is to implement self-adaptation at this level instead

of in the software system itself. Therefore, models work both as information sources that guide adaptation decisions as well as the subjects of adaptation.

The previous waves clarified the requirements of the adaptation manager but the fourth wave stressed the requirements of the software system itself [2]. Examples of these kinds of requirements would be, for example, to keep response time under a specific value or never failing a certain action. The contributions of the fourth wave included different languages for specifying these requirements and communicating them to the adaptation manager.

In the fifth wave, the question was how to guarantee requirements are met in unpredictable runtime environments [2], which is essential in critical self-adaptive systems, with strict requirements. Techniques based on mathematics and statistics such as RQV (Runtime Quantitative Verification) were developed to tackle this problem. Another solution is ActivFORMS (Active FORmal Models for Self-adaptation) that is based on formally guaranteeable models.

The sixth wave also focuses on handling uncertainty but sees control theory as the solution [2]. SimCA* and PBM (Push-Button Methodology) were some of the main contributions of this wave. The study does not discuss learning-based and awareness-based approaches but predicts artificial intelligence, including machine learning, may play a big part in future self-adaptive research for its fundamental ability to handle uncertainty.

5 Conclusion

We studied four general classes of self-adaptive approaches and one specific example from each class. In architecture-based approaches, the focus is on the architectural components of self-adaptive systems and the principles of software engineering that help build them. The more formal control-based approaches apply control theory to software systems in order to formally guarantee self-adaptivity occurs as it should in systems with strict requirements. In learning-based approaches, the traditional MAPE-K architecture is often present but self-adaptivity is improved by allowing the self-adaptive logic itself to change at runtime with incremental learning algorithms. Awareness-based approaches mix aspects of both architecture-based and learning-based approaches while offering improvements such as different types of knowledge and the ability to predict future events.

In the future, it would be important to evaluate and compare self-adaptive approaches more thoroughly, especially in real-world situations. We also left out some approaches, for example those that are architecture-based but still provide formal guarantees for adaptivity. However, based on our results on the benefits and problems of each approach, it seems that an optimal, general self-adaptive solution would combine aspects of all of the studied approaches. In particular, it could be fruitful to base the self-adaptive architecture on the psychological structure of self-awareness and -expression and implement both learning and control processes at the lower levels of this architecture.

References

1. P. Oreizy, M. M. Gorlick, R. N. Taylor, D. Heimbigner, G. Johnson, N. Medvidovic, A. Quilici, D. S. Rosenblum, and A. L. Wolf, "An architecture-based approach to self-adaptive software," *IEEE Intelligent Systems*, vol. 14, no. 3, pp. 54–62, May 1999.
2. P. R. Lewis, A. Chandra, S. Parsons, E. Robinson, K. Glette, R. Bahsoon, J. Torresen, and X. Yao, "A survey of self-awareness and its application in computing systems," in *2011 Fifth IEEE Conference on Self-Adaptive and Self-Organizing Systems Workshops*, Oct 2011, pp. 102–107.
3. T. Chen and R. Bahsoon, "Self-adaptive and online qos modeling for cloud-based software services," *IEEE Transactions on Software Engineering*, vol. 43, no. 5, pp. 453–475, May 2017.
4. R. Van Solingen, V. Basili, G. Caldiera, and H. D. Rombach, "Goal question metric (gqm) approach," *Encyclopedia of software engineering*, 2002.
5. J. O. Kephart and D. M. Chess, "The vision of autonomic computing," *Computer*, vol. 36, no. 1, pp. 41–50, Jan 2003.
6. J. Kramer and J. Magee, "Self-managed systems: An architectural challenge," in *2007 Future of Software Engineering*. IEEE Computer Society, 2007, pp. 259–268.
7. S. Shevtsov, M. Berekmeri, D. Weyns, and M. Maggio, "Control-theoretical software adaptation: A systematic literature review," *IEEE Transactions on Software Engineering*, vol. PP, no. 99, pp. 1–1, 2017.
8. S. Shevtsov, D. Weyns, and M. Maggio, "Handling new and changing requirements with guarantees in self-adaptive systems using simca," in *2017 IEEE/ACM 12th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)*, May 2017, pp. 12–23.
9. A. Elkhodary, N. Esfahani, and S. Malek, "Fusion: A framework for engineering self-tuning self-adaptive software systems," in *Proceedings of the Eighteenth ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, 2010, pp. 7–16.
10. P. R. Lewis, "Self-aware computing systems: From psychology to engineering," in *Design, Automation and Test in Europe Conference and Exhibition*, March 2017, pp. 1044–1049.
11. F. Faniyi, P. R. Lewis, R. Bahsoon, and X. Yao, "Architecting self-aware software systems," in *2014 IEEE/IFIP Conference on Software Architecture*, April 2014, pp. 91–94.
12. U. Neisser, "The roots of self-knowledge: Perceiving self, it, and thou," *Annals of the New York Academy of Sciences*, vol. 818, no. 1, pp. 19–33, 1997.